Naval Research Laboratory

Washington, DC 20375-5000 NRL Report 9019 December 17, 1986

Developing a Software Engineering Methodology for Knowledge-Based Systems

ROBERT J. K. JACOB AND JUDITH N. FROSCHER

Computer Science and Systems Branch
Information Technology Division

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 9019	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b. OFFICE SYMBOL (*If applicable*) Code 7596	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (*City, State, and ZIP Code*) Washington, DC 20375-5000	7b. ADDRESS (*City, State, and ZIP Code*)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61158N; PROJECT NO.; TASK NO. RR014-09-42; WORK UNIT ACCESSION NO. DN490-646

11. TITLE (*Include Security Classification*)
Developing a Software Engineering Methodology for Knowledge-Based Systems

12. PERSONAL AUTHOR(S)
Jacob, Robert J. K. and Froscher, Judith N.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Interim	FROM 1/84 TO 6/86	1986 December 17	28

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Expert systems; Software engineering; Knowledge-based systems; Programming language; Rule-based systems

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Current expert systems are typically difficult to change once they are built. This report describes a design methodology intended to make a knowledge-based system easier to change, particularly by people other than its original developer. The basic approach for solving this problem is to divide the information in a knowledge base and attempt to reduce the amount of information that each single knowledge engineer must understand before he can make a change to the knowledge base. The method thus divides the domain knowledge in an expert system into *groups* and then attempts to limit carefully and specify formally the flow of information between these groups to localize the effects of typical changes within the groups.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Robert J. K. Jacob	22b. TELEPHONE (*Include Area Code*) (202) 767-3365; 22c. OFFICE SYMBOL Code 7596

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

DEVELOPING A SOFTWARE ENGINEERING METHODOLOGY FOR KNOWLEDGE-BASED SYSTEMS

INTRODUCTION

If expert systems are to come into wide use in practical applications, the problem of continuing maintenance and modification of the knowledge base must be addressed in their development. Most current expert systems began as research tools, often developed in universities and maintained by their originators and their students. Many observers now believe that this technology shows promise of solving practical problems in industry and government, but knowledge-based systems continue to be *ad hoc*, one of a kind, and difficult to maintain. Changing a knowledge base typically requires a knowledge engineer who is well-grounded in the design of the system and the structure of the knowledge base. Most often, it requires the same knowledge engineer who originally wrote the system.

This study develops a design and programming methodology to be used by the builder and maintainer of a knowledge-based system. The user of interest in this case is thus the programmer or knowledge engineer who develops or modifies a knowledge base, rather than the end user who obtains information or advice from the expert system. The result is a design technique similar to those used in software engineering [1, 2] that makes a knowledge-based production system easier to change, particularly by people other than its original developer. It provides the developer and maintainer an interface that allows them to organize the information in the system to make it easier to change. We have chosen to concentrate on production systems because they are the most widely used type of knowledge representation in expert systems, particularly among those existing systems that are large enough and mature enough to have experienced the types of maintenance problems we hope to alleviate. In the future, we will attempt to extend the approach to suit other, newer knowledge representations.

BACKGROUND

Few knowledge-based systems are currently being used in commercial or military environments, but one of the most successful exceptions is R1/XCON, developed by McDermott at Carnegie-Mellon University [3]. R1 was designed to configure the many components that make up a DEC VAX 11/780 computer and is implemented as a production system in OPS5. The development history of R1 illustrates the problems encountered when an expert system is used in a practical setting. Since the VAX computer line was constantly being changed and expanded, it was continually necessary to add more knowledge and greater capability to R1. Before long, the system became complex enough that it was necessary to reimplement it entirely. To support R1, DEC established a special software support group that had to invest considerable effort in understanding R1 before they could make any changes to it.

As commercial promise for expert system technology grows, more researchers have become concerned with the practical problems of building these systems for end users who have a limited background in artificial intelligence. They have proposed several methods for partitioning these systems to make the knowledge bases more understandable, easier to maintain, more efficient, and more suitable for parallel processing. One simple approach is to build a system made up of several knowledge bases; examples of this approach are seen in ACE [4] and PROSPECTOR [5]. The developers of LOOPS [6] use the notion of a rule set, which is called like a subroutine. Each rule set returns a single value that can then be used elsewhere in the knowledge base. Clancey has abstracted the inference goals in a

Manuscript approved September 2, 1986.

knowledge base to separate the control strategy, encoded as meta-rules, from the specific domain knowledge [7]. Rules that contribute to a particular goal in a knowledge base can then be grouped; in fact, R1/XCON is partitioned into several subtasks in this fashion [8]. Because expert systems generally use large amounts of computer resources, researchers are studying how both the knowledge base and the working memory can be separated so that each independent group can be processed on a parallel processor [9]. Such a separation can also make the system easier to change, although this was not the original purpose of the study.

APPROACH

The basic approach we have taken for building maintainability into an expert system is to divide the information in the knowledge base and attempt to reduce the amount of information that each single knowledge engineer must understand before he or she can make a change to the knowledge base. We thus divide the domain knowledge in an expert system into *groups* and then attempt to limit carefully and specify formally the flow of information between these groups, to localize the effects of typical changes within the groups.

Production systems comprise a knowledge base, expressed as if-then rules, and a relatively simple inference mechanism or rule interpreter. The interpreter tests the values of the facts on the left-hand side of a rule; if the test succeeds, new values for facts are set according to the right-hand side of the rule. In the present approach, we divide these rules into separate groups. When grouping two rules together, the question to be considered is: *If a change is made to one rule, to what extent is the other rule affected?* In this study, a *fact* refers to some isolable portion of the data representation that, if changed by one rule, affects another rule in some way. In a simple production system where the data are attribute-value pairs, a fact corresponds to one attribute. The knowledge engineer building the system groups together rules that use or produce values for the same sets of facts. With this arrangement, a fact in the knowledge base can be characterized either as being generated and used by rules entirely within a single rule group or else as spanning two or more groups. Facts of the latter variety are critical for future changes to the knowledge base, since these facts are the "glue" that holds the groups together.

Whenever rules in one group use facts generated by rules in other groups, such facts are flagged so that the knowledge engineer will know that their values may have been set outside this group. More importantly, those facts produced by one group and used by rules in other groups must be flagged also. For each such fact, the programmer of the group that produces the fact makes an assertion comprising a brief summary of the information represented by that fact. This assertion is the only information about that fact that should be relied on by the programmers of other groups that use the fact. It is not a formal specification of the information represented by the fact, but rather an informal summary of what the fact should "mean" to outside users.

Given this structure, a programmer who wants to make changes to the system assumes the responsibility of understanding thoroughly and preserving the correct workings of a single group of rules (but not the entire body of rules, as with conventional systems). The programmer is free to make changes to the rules in the group, provided only that he or she preserves the validity of the assertions associated with any facts that are produced by the group and used by other groups. Similarly, whenever the programmer uses a fact that was produced by another group, he relies only on the assertion provided for it by the programmer of the other group, not on any specific information about the fact that might be obtainable from examining the inner workings of the other group.

PROGRAMMING METHODOLOGY

To apply the proposed method, the developer of a rule-based system first divides the rules into groups. This can be done manually or automatically, as described below. One approach is to apply the automatic grouping algorithm to the initial prototype expert system and then use the resulting grouping to guide the organization and development of the final production version.

Next, a software tool is used to characterize each fact as being produced and used entirely within one group (a local or intragroup fact) or being produced or used by two or more groups (an intergroup fact); the latter are flagged. The developer of each rule group that produces intergroup facts then provides an assertion describing each such fact. The assertion describes what the programmer of the group that produces the fact asserts will be true concerning that fact, upon which the programmers of the groups using that fact can rely. It thus summarizes the workings of the group that produces it. It is generally inappropriate for this assertion to provide a formal specification of the conditions for producing the fact, because this would essentially repeat the entire set of rules as they are presently. Rather, what is desired is an informal statement of the aspects of the output that will not change and may be considered externally visible. For example: *Fact X gives the system's best estimate of whether the patient has heart disease*; rather than: *X will be true if input* $A < 0.6$ *and* $B \geqslant 2.1$. This description is the only information about the fact that should be used in the development of other groups containing rules that use the value of the fact; information about the internal workings of the rules in the group should not be used, because such details are not guaranteed to remain unchanged.

Facts produced simultaneously by a large number of groups are not handled in precisely this way, since they constitute the "global" data for the system. Their assertions are part of the overall system design, rather than the design of the individual groups. Logically, the assertion for such a fact could be the "or" of the assertions generated by all groups that produce the fact, but this would not be very useful. Instead, the overall system designer will identify these facts in the early design and provide more abstract statements, that indicate how he expects the global data to be used by all the groups together. A well-designed system will minimize the number of facts of this kind.

Thus, the set of rules will be divided into groups, the intergroup facts used and produced by each group will be identified, and descriptions will be entered for those produced by each group. The example below illustrates the language used to provide this information, using a simple example knowledge base [10].

```
(GROUP isamammal
  (PRODUCES
    (mammal "is it a mammal, by conventional English usage"))
  (RULES
     (r1 (IF hair) (THEN mammal))
     (r2 (IF milk) (THEN mammal))))

(GROUP isabird
  (PRODUCES
    (bird "is it a bird, by English usage"))
  (RULES
     (r3 (IF feather) (THEN bird))
     (r4 (IF flies ovip) (THEN bird))

     (rx (IF a) (THEN c))
     (ry (IF b) (THEN c))
     (rz (IF c) (THEN bird))))

(GROUP isacarn
  (PRODUCES
    (carn "is it a carnivorous creature"))
  (RULES
     (r5 (IF meat) (THEN carn))
     (r6 (IF pointed claws fwdeyes) (THEN carn))))
```

```
(GROUP isungulate
  (PRODUCES
    (ungulate "is it an ungulate"))
  (USES
     (mammal))
  (RULES
     (r7 (IF mammal hoofs) (THEN ungulate))))

(GROUP kind-of-carn
  (USES
     (mammal)
     (carn))
  (RULES
     (r8 (IF mammal carn tawny darksp) (THEN cheetah))
     (r9 (IF mammal carn tawny blackst) (THEN tiger))))

(GROUP kind-of-ungulate
  (USES
     (ungulate))
  (RULES
     (r10 (IF ungulate longn longl darksp) (THEN giraffe))
     (r11 (IF ungulate blackst) (THEN zebra))))

(GROUP kind-of-bird
  (USES
     (bird))
  (RULES
     (r12 (IF bird notfly longn longl blackwh) (THEN ostrich))
     (r13 (IF bird notfly swims blackwh) (THEN penguin))
     (r14 (IF bird flyswell) (THEN albatross))))
```

The rules above are given in an abstract notation that lists their input and output facts, but no further details. This is the same representation used as input for the analyses described subsequently.

Note that three extra rules have been added for illustration to group `isabird` above. They indicate that the animal is considered a bird if `c` is true, which in turn depends on `a` and `b`. These rules appear to produce fact `c`, but `c` is not shown in the `PRODUCES` list for group `isabird` and does not have an assertion. This is because `c` is not used by rules in any other group; it is thus an intragroup fact, analogous to a local variable with no bearing on the connectivity of the modules of the system. Also, group `kind-of-carn` appears to produce facts `cheetah` and `tiger`, but they, too, are not listed. The reason is that these facts are not *used* by rules in any other groups and, again, have no effect on the connections between groups. In fact, they are top-level outputs of this expert system.

Some of the steps required to produce a knowledge base like the above are manual, and some can be automated. To prepare this example, the rules were first manually divided into groups (although this could have been done automatically, as described below). Then, a software tool was used to analyze the groups and automatically list the intergroup facts produced and used by each group. The programmer then manually wrote the (here, trivial) assertions for all intergroup facts produced by each group. This last step is not automatic and is critical to the success of the method. The methodology and tools find and highlight the intergroup facts and then require the programmer to give them special attention and provide special declarations for them.

Finally, note in the above example that all information except for the rules themselves—i.e., the groupings, `PRODUCES` and `USES` lists, and assertions—are essentially comments. They need have no effect on the actual execution (or efficiency) of the expert system.

After a knowledge base is developed in this fashion, the knowledge engineer who wants to modify a group must understand the internal operations of that group, but not of the rest of the knowledge base. If he or she preserves the correct functioning of the rules within the group and does not change the validity of the assertions about its intergroup facts, the knowledge engineer can be confident that the change that has been made will not adversely affect the rest of the system. Conversely, if the knowledge engineer wants to use additional intergroup facts from other groups, he or she should rely only on the assertions provided for them, not on the internal workings of the rules in the other group. (Of course, changes that pervade several groups still have to be handled as they always have been, but the grouping is intended to minimize these.)

An interesting aspect of this approach is that it draws distinctions between the facts contained in the working memory of a production system. Certain facts are flagged as being important to the overall software structure of the system, while others are "internal" to particular modules and thus less important. Programmers are advised to pay special attention to rules that involve the "important" facts. This is in contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, where they must all command equal attention or inattention from the programmer.

ALGORITHMS FOR PARTITIONING A KNOWLEDGE BASE

It is possible to develop an algorithm that will take a set of rules and divide them into groups suitable for use with this method. Such an algorithm can be used to help develop a rule-based system, and it will also be useful in evaluating the general applicability of the proposed method by grouping the rules of existing systems, as described subsequently. It is somewhat easier to develop a grouping algorithm for rule-based systems than for code written in conventional programming languages because both the syntax and semantics of production languages are quite simple and regular. (Some refinements needed to handle OPS5, a relatively complex production language, are discussed below.)

The basic problem is to partition a set of rules into groups that will aid in their maintenance. Rules that affect each other and are likely to be changed at the same time should be grouped. Several approaches to the grouping problem have been explored [11, 12], and the best results are obtained from an algorithm based on cluster analysis. Given a collection of objects, a clustering algorithm partitions them into groups of like objects. To use such an algorithm, though, a measure of distance or "relatedness" between the objects (rules) must be defined. Since our ultimate concern is for a programmer making changes to the knowledge base, this similarity between two rules should measure the likelihood that a change made to one rule would require a change in the other rule.

The rules in a production system are connected or related to each other through the facts whose values they use or modify. These relationships can be depicted in a graph, showing the inference hierarchy for the system. Figure 1 shows the animal system given above. In it, each node, or circle, represents a rule, and each link, or line, between two rules represents a fact whose value is set by one rule and used by the other. A simple measure of the "relatedness" of two rules is the number of facts that are mentioned in both rules. Since there are several ways in which two rules could refer to the same fact, we decided to weight this count. The two rules

```
if A then B
if B then C
```

share fact B in common; so do the two rules

```
if A then B
if C then B .
```

The rules of the former pair seem to have a greater programming effect on each other than the latter pair, and hence should be more "related." Figure 2 summarizes the three ways in which two rules can

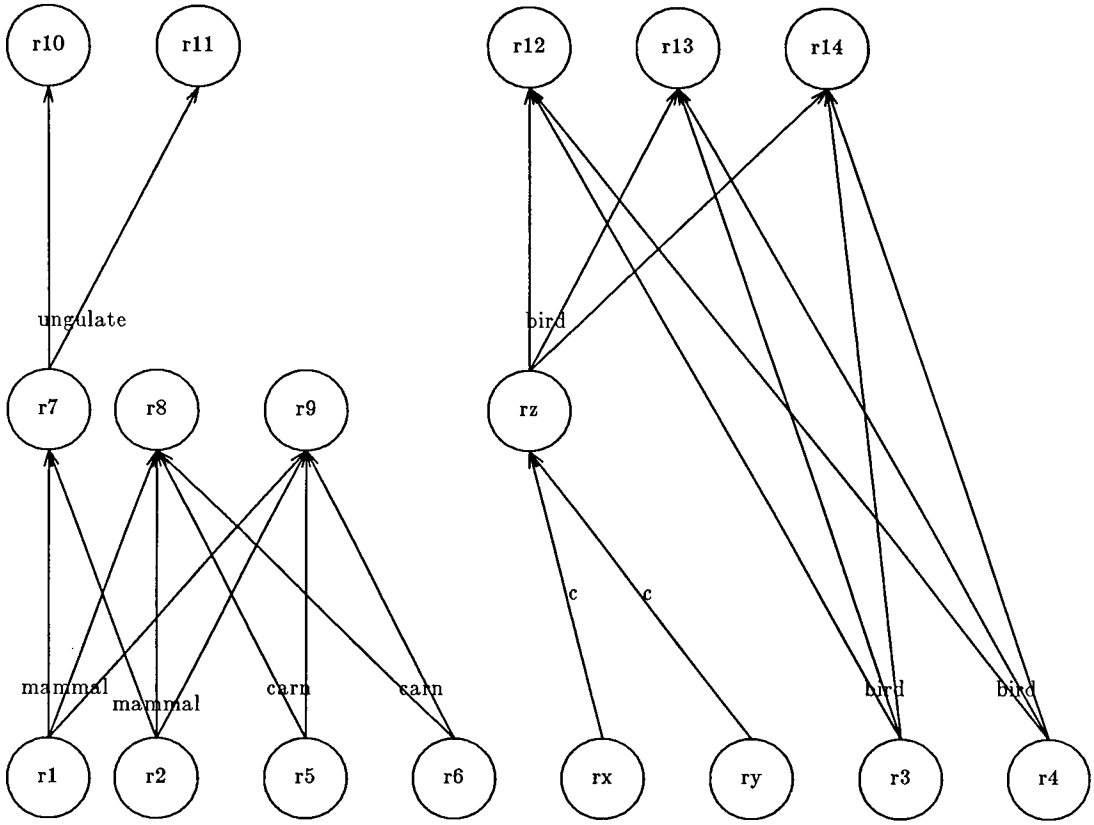
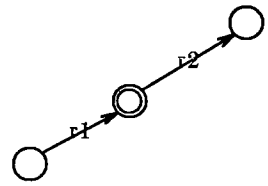
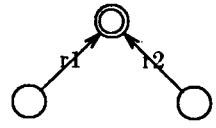


Fig. 1 — Plot of individual rules of a very simple expert system

Score(r1,r2) = 1.0



Score(r1,r2) = 0.75



Score(r1,r2) = 0.5

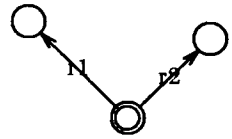


Fig. 2 — Components of "relatedness" measure between two rules

share a fact and the weight given to each. The total "relatedness" measure between two rules is, then, a weighted count of the facts shared by both rules, where each fact is weighted by the score that indicates in which of the three possible ways the two rules use the fact.

Given such a measure, we proceed with a straightforward clustering algorithm. First, measure the similarities between all pairs of rules, select the closest pair, and put those two rules together into one cluster. Then, repeat the procedure, grouping rules with each other or possibly with already-formed clusters. In the latter case, we must measure the "relatedness" between a rule and a cluster of rules. This is simply defined as the mean of the (square roots of the) similarities between the individual rule and each of the rules in the cluster, corresponding to an average-linkage clustering procedure. The algorithm proceeds iteratively.

The algorithm presently used contains some further refinements. While they have a relatively small effect on its operation, they are significant at iterations where the best and next-best possible pairings are close in average similarity but differ substantially on other criteria. First, other values of the weights for the three cases shown in Fig. 2 were tested, and the clusterings obtained were generally insensitive to any choice of plausible weights. Then, an additional case was added to the three shown; this covers facts *not* shared by two rules. Without it, the pair of rules

```
if A then B
if B then C
```

and the pair

```
if A then B
if B then C and D and E and F
```

would have the same similarity; but, in determining a grouping, the former pair is preferable to the latter. Hence, a penalty is added to the similarity between two rules for each fact that appears in one of the rules but not the other. Such facts are weighted by -0.25 and added to the similarity measure shown in Fig. 2. This reduces the tendency of the algorithm to use a few pivotal rules to form a long, "spindly" group that gradually annexes its neighbors in all directions. The other change is an additional term added to the similarity between two groups that slightly favors combinations of smaller groups over larger ones. This takes into account the practical consideration that the groupings are ultimately going to be used as the basis for assigning programming responsibilities to individuals, hence very small or very large groups are likely the be manually overridden in the end. For this purpose, the quantity *2/(size of the combined group)* is added to the similarity between two groups.

The clustering algorithm described makes no special provision for state or goal variables in the rule set. They are treated exactly as any other facts. While one could use the state variables as a key to identify groupings of rules, such a method will only divide the rules temporally, that is, according to *when* they operate. This is not always the appropriate grouping from the point of view of modifying the program. For modification, we are concerned with how two rules affect each other in the long run, not whether they affect each other in sequence. The two characteristics are often—but not guaranteed to be—the same, in well-structured systems. Thus, the clustering algorithm treats the state variables like all other variables in identifying groups. In the particular system analyzed below, it is seen that, by following this approach, the algorithm "discovered" the same grouping as that implied by the state variables. It has also worked well for systems that do not use state variables.

One drawback to algorithms of this type is that on each iteration the algorithm makes the best possible agglomeration of two groups, but it never backtracks, in case there might be a better grouping for the system considered as a whole. Also, like most clustering algorithms, if it runs for enough iterations it will eventually group all the rules into one large group. One stopping rule that has worked in several cases is to stop when the similarity between the next two groups to be combined is no longer positive (since there is a term with a negative weight in the similarity between rules). Another approach is to use the distribution of the sizes of the groups as a guide.

AN EXAMPLE

A somewhat larger rule-based system, developed by Ralph Fink et al. of the Naval Air Development Center, is shown in the example in this section. It has 34 rules and 21 facts. This system will illustrate the programming method, but it should be remembered that it is used for exposition; the proposed method will seem excessive for a system of this size and relative simplicity. The system is written in the OPS5 language [13, 14] and describes the interactions between two opposing military battle groups. The original system, in OPS5, is shown below. Some details in the bodies of the rules, such as write statements and some literal values have been deleted or replaced with ellipses to save space.

```
(p goal-rule-1
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal attack)
  (remove <goal-rules>)
  (make plan-rules ^status active))

(p goal-rule-2
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal attack)
  (remove <goal-rules>)
  (make revision-rules ^status active))

(p goal-rule-3
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal attack)
  (remove <goal-rules>)
  (make revision-rules ^status active))

(p goal-rule-4
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal attack)
  (remove <goal-rules>)
  (make revision-rules ^status active))

(p goal-rule-5
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal advance)
  (remove <goal-rules>)
  (make plan-rules ^status active))

(p goal-rule-6
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal no-attack)
  (remove <goal-rules>)
  (make plan-rules ^status active))
```

```
(p goal-rule-7
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position ... ^relative-behavior ...) <target>}
  -->
  (modify <target> ^goal retreat)
  (remove <goal-rules>)
  (make revision-rules ^status active))

(p goal-rule-8
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position arrived-at) <target>}
  -->
  (make battle-group ^status dead)
  (modify <target> ^goal achieved)
  (remove <goal-rules>)
  (make plan-rules ^status active))

(p goal-rule-9
  {(goal-rules ^status active) <goal-rules>}
  {(target ^relative-behavior inconsistent) <target>}
  -->
  (modify <target> ^goal dont-know)
  (remove <goal-rules>)
  (make plan-rules ^status active))

(p plan-rule-1
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal attack) <target>}
  -->
  (modify <target> ^plan-of-action course-to)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(p plan-rule-2
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal no-attack) <target>}
  -->
  (modify <target> ^plan-of-action no-course-to)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(p plan-rule-3
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal dont-know) <target>}
  -->
  (modify <target> ^plan-of-action wait-and-see)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(p plan-rule-4
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal achieved) <target>}
  -->
  (modify <target> ^plan-of-action succeeded)
  (remove <plan-rules>)
  (make expectation-rules ^status active))
```

```
(p plan-rule-5
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal advance) <target>}
  -->
  (modify <target> ^plan-of-action course-to)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(p revision-rule-1
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ... ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action feint)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(p revision-rule-2
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^goal retreat) <target>}
  -->
  (modify <target> ^plan-of-action no-course-to)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(p revision-rule-3
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ... ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action feint)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(p revision-rule-4
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ... ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action wait-and-see)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(p expectation-rule-1
  {(expectation-rules ^status active) <expectation-rules>}
  {(target ^plan-of-action course-to) <target>}
  -->
  (make computational-rules ^status active)
  (remove <expectation-rules>)
  ;; Note: update-attributes is an external function, which
  ;; sets the transition and present-position attributes of target
  (modify <target> (update-attributes)))

(p expectation-rule-2
  {(expectation-rules ^status active) <expectation-rules>}
  {(target ^plan-of-action no-course-to) <target>}
  -->
  (make computational-rules ^status active)
  (remove <expectation-rules>)
  (modify <target> (update-attributes)))
```

```
(p expectation-rule-3
  {(expectation-rules ^status active) <expectation-rules>}
  {(target ^plan-of-action feint) <target>}
  -->
  (make computational-rules ^status active)
  (remove <expectation-rules>)
  (modify <target> (update-attributes)))

(p expectation-rule-4
  {(expectation-rules ^status active) <expectation-rules>}
  {(target ^plan-of-action wait-and-see) <target>}
  -->
  (make computational-rules ^status active)
  (remove <expectation-rules>)
  (modify <target> (update-attributes)))

(p expectation-rule-5
  {(expectation-rules ^status active) <expectation-rules>}
  {(target ^plan-of-action succeeded) <target>}
  -->
  (halt))

(p computation-rule-1
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition recede) <target>}
  -->
  (modify <target> ^relative-behavior 1-move-away)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-2
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition approach) <target>}
  -->
  (modify <target> ^relative-behavior moved-towards)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-3
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition recede) <target>}
  -->
  (modify <target> ^relative-behavior 2-moves-away)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-4
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition recede) <target>}
  -->
  (modify <target> ^relative-behavior 3-moves-away)
  (remove <computational-rules>)
  (make goal-rules ^status active))
```

```
(p computation-rule-5
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition recede) <target>}
  -->
  (modify <target> ^relative-behavior 4-moves-away)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-6
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior ... ^transition approach) <target>}
  -->
  (modify <target> ^relative-behavior reapproach)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-7
  {(computational-rules ^status active) <computational-rules>}
  {(target ^relative-behavior reapproach ^transition recede) <target>}
  -->
  (modify <target> ^relative-behavior inconsistent)
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-8
  {(computational-rules ^status active) <computational-rules>}
  {(target ^transition none) <target>}
  -->
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p computation-rule-9
  {(computational-rules ^status active) <computational-rules>}
  -->
  (remove <computational-rules>)
  (make goal-rules ^status active))

(p starter-rule-1
  {(first-starter-rule ^status active) <first-starter-rule>}
  {(target ^present-position none) <target>}
  -->
  (remove <first-starter-rule>)
  ;; Note: initial-position sets the present-position attribute of target
  (modify <target> (initial-position)))

(p starter-rule-2
  {(other-starter-rules ^status active) <other-starter-rules>}
  {(target ^present-position ...) <target>}
  -->
  (remove <other-starter-rules>)
  (modify <target> (update-attributes))
  (make computational-rules ^status active))
```

This is an unusually well-structured rule-based system, and it makes heavy use of state variables (the `status` attributes of `goal-rules`, `plan-rules`, etc.) These variables are used to enable or disable sets of rules. The system thus already contains an implicit structure; for example, all rules that depend on the left-hand side

```
(goal-rules ^status active)
```

will be enabled or disabled together and constitute a logical grouping to the original system developer as well as a temporal grouping.

Using the new method, this set of rules would appear as shown below, with the rules divided into groups and the intergroup facts declared. "Facts" in OPS5 are not always simple static variables; the next section explains how they are interpreted in this framework. The division into groups shown below was obtained from the clustering algorithm. Note that the groupings are similar to those implied by the state variables; that is, the clustering algorithm "discovered" the structure already implicit in this rather well-structured set of rules.

```
(GROUP computation-rules

  (PRODUCES
    (target-relative-behavior
      "Describes direction of target with respect to current position
      and how many moves it would take to reach the target.
      Possible values are:
      arrived-at, moved-towards, 1-move-away, 2-moves-away,
      3-moves-away, 4-moves-away, reapproach, and inconsistent.")

    (goal-rules-status
      "State variable, enables firing of goal rules"))

  (USES
     (target-relative-behavior)
     (target-transition)
     (computational-rules-status))

  (RULES
     (computation-rule-1
       {(computational-rules ^status active) <computational-rules>}
       {(target ^relative-behavior ... ^transition recede) <target>}
       -->
       (modify <target> ^relative-behavior 1-move-away)
       (remove <computational-rules>)
       (make goal-rules ^status active))

     (computation-rule-2
       {(computational-rules ^status active) <computational-rules>}
       {(target ^relative-behavior ... ^transition approach) <target>}
       -->
       (modify <target> ^relative-behavior moved-towards)
       (remove <computational-rules>)
       (make goal-rules ^status active))

     (computation-rule-3
       {(computational-rules ^status active) <computational-rules>}
       {(target ^relative-behavior ... ^transition recede) <target>}
       -->
       (modify <target> ^relative-behavior 2-moves-away)
       (remove <computational-rules>)
       (make goal-rules ^status active))
```

```
    (computation-rule-4
      {(computational-rules ^status active) <computational-rules>}
      {(target ^relative-behavior ... ^transition recede) <target>}
      -->
      (modify <target> ^relative-behavior 3-moves-away)
      (remove <computational-rules>)
      (make goal-rules ^status active))

    (computation-rule-5
      {(computational-rules ^status active) <computational-rules>}
      {(target ^relative-behavior ... ^transition recede) <target>}
      -->
      (modify <target> ^relative-behavior 4-moves-away)
      (remove <computational-rules>)
      (make goal-rules ^status active))

    (computation-rule-6
      {(computational-rules ^status active) <computational-rules>}
      {(target ^relative-behavior ... ^transition approach) <target>}
      -->
      (modify <target> ^relative-behavior reapproach)
      (remove <computational-rules>)
      (make goal-rules ^status active))

    (computation-rule-7
      {(computational-rules ^status active) <computational-rules>}
      {(target ^relative-behavior reapproach ^transition recede) <target>}
      -->
      (modify <target> ^relative-behavior inconsistent)
      (remove <computational-rules>)
      (make goal-rules ^status active))

    (computation-rule-8
      {(computational-rules ^status active) <computational-rules>}
      {(target ^transition none) <target>}
      -->
      (remove <computational-rules>)
      (make goal-rules ^status active))

    (computation-rule-9
      {(computational-rules ^status active) <computational-rules>}
      -->
      (remove <computational-rules>)
      (make goal-rules ^status active))))

(GROUP expectation-plan-revision

  (PRODUCES
    (target-transition
      "Tells whether target is currently receding or approaching")

    (target-present-position
      "Tells which of 3 possible sectors the target currently occupies")

    (computational-rules-status
      "State variable, enables firing of computational-rules"))
```

```
(USES
   (target-goal)
   (target-present-position)
   (target-relative-behavior)
   (plan-rules-status)
   (revision-rules-status))

(RULES
   (expectation-rule-1
     {(expectation-rules ^status active) <expectation-rules>}
     {(target ^plan-of-action course-to) <target>}
     -->
     (make computational-rules ^status active)
     (remove <expectation-rules>)
     (modify <target> (update-attributes)))

   (expectation-rule-2
     {(expectation-rules ^status active) <expectation-rules>}
     {(target ^plan-of-action no-course-to) <target>}
     -->
     (make computational-rules ^status active)
     (remove <expectation-rules>)
     (modify <target> (update-attributes)))

   (expectation-rule-3
     {(expectation-rules ^status active) <expectation-rules>}
     {(target ^plan-of-action feint) <target>}
     -->
     (make computational-rules ^status active)
     (remove <expectation-rules>)
     (modify <target> (update-attributes)))

   (expectation-rule-4
     {(expectation-rules ^status active) <expectation-rules>}
     {(target ^plan-of-action wait-and-see) <target>}
     -->
     (make computational-rules ^status active)
     (remove <expectation-rules>)
     (modify <target> (update-attributes)))

   (expectation-rule-5
     {(expectation-rules ^status active) <expectation-rules>}
     {(target ^plan-of-action succeeded) <target>}
     -->
     (halt))

   (plan-rule-1
     {(plan-rules ^status active) <plan-rules>}
     {(target ^goal attack) <target>}
     -->
     (modify <target> ^plan-of-action course-to)
     (remove <plan-rules>)
     (make expectation-rules ^status active))

   (plan-rule-2
     {(plan-rules ^status active) <plan-rules>}
     {(target ^goal no-attack) <target>}
     -->
     (modify <target> ^plan-of-action no-course-to)
     (remove <plan-rules>)
     (make expectation-rules ^status active))
```

```
(plan-rule-3
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal dont-know) <target>}
  -->
  (modify <target> ^plan-of-action wait-and-see)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(plan-rule-4
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal achieved) <target>}
  -->
  (modify <target> ^plan-of-action succeeded)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(plan-rule-5
  {(plan-rules ^status active) <plan-rules>}
  {(target ^goal advance) <target>}
  -->
  (modify <target> ^plan-of-action course-to)
  (remove <plan-rules>)
  (make expectation-rules ^status active))

(revision-rule-1
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ...
       ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action feint)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(revision-rule-2
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^goal retreat) <target>}
  -->
  (modify <target> ^plan-of-action no-course-to)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(revision-rule-3
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ...
          ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action feint)
  (remove <revision-rules>)
  (make expectation-rules ^status active))

(revision-rule-4
  {(revision-rules ^status active) <revision-rules>}
  {(target ^relative-behavior ... ^present-position ...
          ^goal ...) <target>}
  -->
  (modify <target> ^plan-of-action wait-and-see)
  (remove <revision-rules>)
  (make expectation-rules ^status active))))
```

```
(GROUP goal-rules

  (PRODUCES
    (target-goal
      "Gives objective currently being sought with respect to target,
      i.e., attack, no-attack, retreat, advance, achieved,
      or dont-know")

    (revision-rules-status
      "State variable, enables firing of revision-rules")

    (plan-rules-status
      "State variable, enables firing of plan-rules"))

  (USES
    (target-present-position)
    (target-relative-behavior)
    (goal-rules-status))

  (RULES
     (goal-rule-1
       {(goal-rules ^status active) <goal-rules>}
       {(target ^present-position ... ^relative-behavior ...) <target>}
       -->
       (modify <target> ^goal attack)
       (remove <goal-rules>)
       (make plan-rules ^status active))

     (goal-rule-2
       {(goal-rules ^status active) <goal-rules>}
       {(target ^present-position ... ^relative-behavior ...) <target>}
       -->
       (modify <target> ^goal attack)
       (remove <goal-rules>)
       (make revision-rules ^status active))

     (goal-rule-3
       {(goal-rules ^status active) <goal-rules>}
       {(target ^present-position ... ^relative-behavior ...) <target>}
       -->
       (modify <target> ^goal attack)
       (remove <goal-rules>)
       (make revision-rules ^status active))

     (goal-rule-4
       {(goal-rules ^status active) <goal-rules>}
       {(target ^present-position ... ^relative-behavior ...) <target>}
       -->
       (modify <target> ^goal attack)
       (remove <goal-rules>)
       (make revision-rules ^status active))

     (goal-rule-5
       {(goal-rules ^status active) <goal-rules>}
       {(target ^present-position ... ^relative-behavior ...) <target>}
       -->
       (modify <target> ^goal advance)
       (remove <goal-rules>)
       (make plan-rules ^status active))
```

```
    (goal-rule-6
      {(goal-rules ^status active) <goal-rules>}
      {(target ^present-position ... ^relative-behavior ...) <target>}
      -->
      (modify <target> ^goal no-attack)
      (remove <goal-rules>)
      (make plan-rules ^status active))

    (goal-rule-7
      {(goal-rules ^status active) <goal-rules>}
      {(target ^present-position ... ^relative-behavior ...) <target>}
      -->
      (modify <target> ^goal retreat)
      (remove <goal-rules>)
      (make revision-rules ^status active))

    (goal-rule-8
      {(goal-rules ^status active) <goal-rules>}
      {(target ^present-position arrived-at) <target>}
      -->
      (make battle-group ^status dead)
      (modify <target> ^goal achieved)
      (remove <goal-rules>)
      (make plan-rules ^status active))

    (goal-rule-9
      {(goal-rules ^status active) <goal-rules>}
      {(target ^relative-behavior inconsistent) <target>}
      -->
      (modify <target> ^goal dont-know)
      (remove <goal-rules>)
      (make plan-rules ^status active))))


(GROUP starter-rules

  (PRODUCES
    (target-transition
      "Tells whether target is currently receding or approaching")

    (target-present-position
      "Tells which of 3 possible sectors the target currently occupies")

    (computational-rules-status
      "State variable, enables firing of computational-rules"))

  (USES
    (target-present-position))

  (RULES

    (starter-rule-1
      {(first-starter-rule ^status active) <first-starter-rule>}
      {(target ^present-position none) <target>}
      -->
      (remove <first-starter-rule>)
      (modify <target> (initial-position)))


    (starter-rule-2
      {(other-starter-rules ^status active) <other-starter-rules>}
      {(target ^present-position ...) <target>}
      -->
      (remove <other-starter-rules>)
      (modify <target> (update-attributes))
      (make computational-rules ^status active))))
```

Examination of the PRODUCES and USES lists gives an indication of the structure of the system. It becomes clear that four of the attributes of class `target` are the only real nonlocal data in the system (the remaining 17 facts are all either local variables or state variables). The method helps to identify and focus attention on these four key variables. In addition to using and producing these four `target` variables, each group is seen to use those state variables that enable its own rules and produce state variables that enable exactly one other group, which indicates a particularly straightforward pattern of activation among these rule groups.

"FACTS" IN WORKING MEMORY ELEMENTS

Several production system languages, in particular OPS5, add some wrinkles to the straightforward conception of static facts examined and modified by rules. A rule in OPS5 can test and set values of one or more attributes of one or more "working memory elements" and can create or remove such elements. The "facts" of interest are thus dynamic—they cannot be statically enumerated in advance. Each working memory element is identified as belonging to some *class*, and rules generally identify the class of each element they examine or change.

For example, the following OPS5 rule

```
(p goal-rule-1
  {(goal-rules ^status active) <goal-rules>}
  {(target ^present-position 123 ^relative-behavior 456) <target>}
  -->
  (modify <target> ^goal attack)
  (remove <goal-rules>)
  (make plan-rules ^status active))
```

examines the `status` attribute of a working memory element of class `goal-rules` and the `present-position` and `relative-behavior` attributes of one of class `target`. It changes the `goal` attribute of the `target` element, removes the `goal-rules` element, and creates a new element of class `plan-rules` and sets its `status` attribute. For the purposes of the software engineering method, each of the possible classes of elements in a system can be treated as a separate "fact," with a multifaceted value consisting of all the attribute values of elements of that class. The above rule would then be considered to use facts `goal-rules` and `target` and produce facts `target`, `goal-rules`, and `plan-rules`. In the notation we use for abstract description of rules, it would be written:

```
(RULE goal-rule-1
  (IF goal-rules target)
  (THEN target goal-rules plan-rules))
```

The problem with this interpretation is that it is a rather coarse-grained abstraction of the working of the rules. For example, nearly every rule in the system shown below tests some attribute of target. A finer-grained abstraction of the information used and produced by each rule is needed. The basic goal of this abstraction is to identify which items in the right-hand sides of rules could affect the firing or not firing of which items in the left-hand sides. Observe that the right-hand side item

```
(modify <target> ^goal attack)
```

in the above does not affect any rule with a left-hand side containing

```
(target ^present-position 789).
```

It affects only those rules that test the `goal` attribute of working memory elements of class `target`. Thus, each attribute of each class is treated as a separate "fact," e.g., `target-goal` and `target-present-position` are separate facts.

Since working memory elements are created and destroyed dynamically, there is not really a single static variable `target-goal` in the system; it is considered to occur whenever a working memory element of class `target` uses an attribute named `goal`. The fact `target-goal` is an abstraction of all possible uses of the `goal` attribute of *all* working memory elements of class `target`.

We observe, similarly, that a right-hand side

```
(make plan-rules ^status active)
```

can only affect rules whose left-hand sides test the `status` attribute of class `plan-rules`, since no other attributes are initialized. Hence, it is considered to set "fact" `plan-rules-status` only. However, the right-hand side

```
(remove <goal-rules>)
```

that deletes an entire working memory element of class `goal-rules` potentially affects any rule that examines any portion of a `goal-rules` element. It is considered to set a new "fact" `goal-rules` that may be thought of as an abstraction for *the effect of removing an element of class* `goal-rules`. All left-hand sides that examine any part of a `goal-rules` working memory element are then considered to use the fact `goal-rules` in addition to the facts for any attributes of `goal-rules` they use.

The OPS5 rule above thus finally becomes, in our representation:

```
(RULE goal-rule-1
  (IF goal-rules goal-rules-status
    target target-present-position target-relative-behavior)
  (THEN target-goal goal-rules plan-rules-status)).
```

As noted, these facts are not individual static variables but abstractions of the manipulations of working memory data by which two rules can affect each other.

SUPPORT SOFTWARE

We have developed software tools to support this programming methodology and to analyze the connections between the rules of a production system. The input is a set of rules expressed in the abstract form shown above. We have built software that translates from OPS5 into this representation, including separating OPS5 working memory elements into their component "facts." For expert systems written in other languages, we have performed the translation manually or semiautomatically with text processing programs.

The developer of a rule-based system can define the grouping of rules and input the knowledge base in the form shown in the previous examples, or he or she can use the clustering algorithm discussed to produce the grouping. Given such a grouping, the software then identifies the intragroup and intergroup facts. It flags all intergroup facts produced by a group so the programmer can provide assertions for them; and it flags all intergroup facts used by a group and retrieves their descriptions so the programmer can rely on them when using such facts.

Other software tools can trace all effects of changing a given rule and can find any unused rules or groups. Statistics about the characterization of the facts in the system are also produced. For example, the following data describe the system shown above:

```
34   Rules
 4   Groups

21   Facts
```

```
3    bottom
1    top
9    x-x
4    x-y
0    x-xy
0    xy-x
0    xy-xy
1    x-any
2    any-x
1    any-any
```

Of the 21 facts found, 3 of them are characterized as **bottom**, meaning input, data not produced by any rules, and 1 as **top**, meaning an output, not used by any rules. The **x-x** category denotes intragroup facts. The remaining categories describe subspecies of intergroup facts: **x-y** denotes facts produced by just one group and used by just one other group; **x-xy** are those produced by one group and used by that group and one other group; **xy-x** are produced by two groups and used only by one of those two; **xy-xy** are produced by two groups and used by the same two; **x-any** are produced by one group and used by two or more groups; **any-x** are produced by two or more groups and used by one group; and **any-any** covers the remaining more complex cases. Finally, software tools are available to compute the measures of coupling and cohesion discussed below.

EVALUATING THE METHODOLOGY

To decide whether partitioning a knowledge base is a feasible approach, we are analyzing existing expert systems to determine how the rules in the system are related to each other. We are using the software tools to determine whether the rules are indeed thoroughly intertwined or sufficiently separated so that they could be divided into the groups required by the methodology. The approach uses the clustering algorithm to divide the rules of the existing system into groups. By examining the resulting groupings, we hope to determine how well the structure implied by the new programming method fit the structures observed in actual rule bases, that is, whether the existing systems *could* have been cast in the mold required by the method or whether it would have imposed excessive restrictions and unnatural structure on the developers. To date, we have analyzed several knowledge bases and found that there is considerable separability and latent structure to the relationships between the rules in these systems. This permits the present approach to be imposed. However, such structure has not been exploited to improve maintainability.

For example, Fig. 3 shows a graph for a larger expert system. It was developed by James Reggia of the University of Maryland, using the KES language [15] and is used to diagnose stroke and related diseases. It contains 373 rules and 116 distinct variables. Unlike OPS5, the KES language uses static variables, in which all instantiations are declared in advance. However, 21 of the variables in this system can be simultaneously assigned more than one value. The presence of each possible value for each multivalued variable is treated as a separate fact in our framework, so that the system has 244 facts. In Fig. 3, like Fig. 1, each node represents a rule, and each link between two rules represents a fact whose value is set by one rule and used by the other. To reduce clutter, labels like those in Fig. 1 have been suppressed, and rules, that were represented by circles in Fig. 1, are now shown as points.

Figure 4 shows the same system as Fig. 3, after clustering into 30 groups. Each node now represents a group of rules, and each link represents a fact that is produced by rules in one group and used by those in another group. Facts that are produced and used entirely within a single group do not appear in the graph, and the resulting structure is clearly simpler. Statistics for this grouping are as follows:

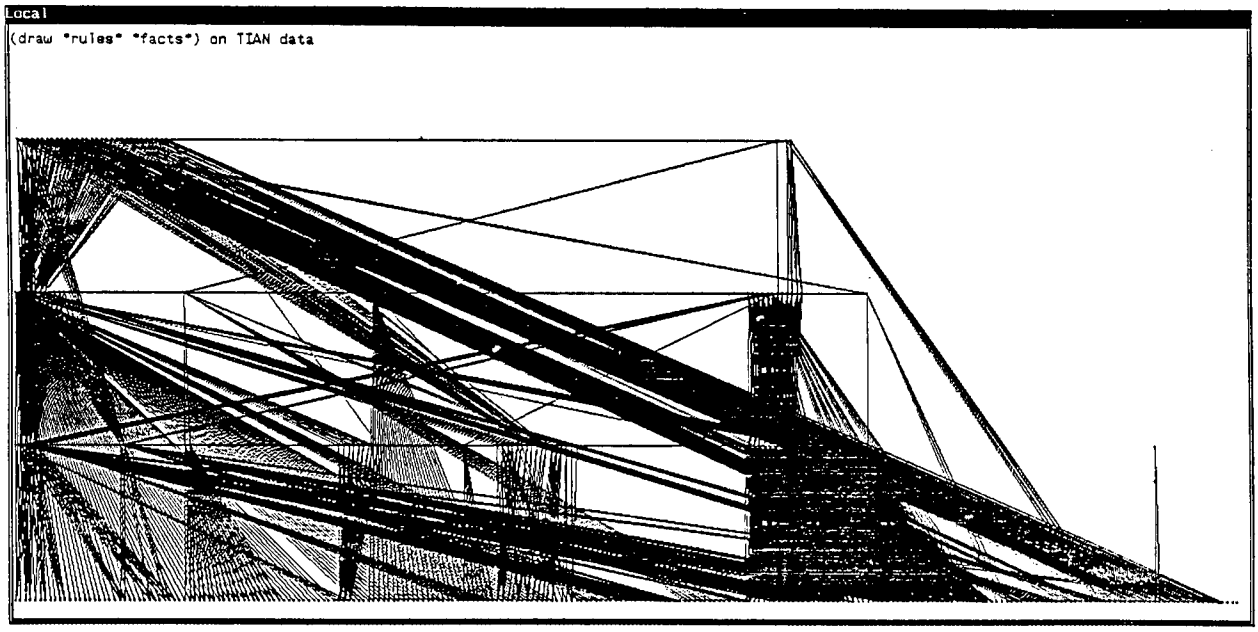


Fig. 3 — Plot of rules of a more complex expert system

```
373   Rules
 30   Groups

244   Facts

 94   bottom
 30   top
 66   x-x
 12   x-y
  6   x-xy
  7   xy-x
  0   xy-xy
 21   x-any
  5   any-x
  3   any-any
```

Observe that 66 of the 120 nontop- or bottom-level facts in this system have become intragroup facts or local variables.

MEASURES OF COUPLING AND COHESION

The division of a set of rules into groups should attempt to minimize the amount of coupling between the groups and maximize the amount of cohesiveness within each group [16]. Defining measures for these notions will provide data to help compare alternative groupings of a given set of rules. Once a set of rules is divided into groups, each fact in the system is characterized as intergroup or intragroup. One simple measure of coupling is the proportion of intergroup facts, while cohesion is represented by the proportion of intragroup facts. For the battle-planning system shown previously, this measure gives 8/17 for coupling and 9/17 for cohesion (top- and bottom-level facts are excluded from this count).

Another approach to these measures is also being investigated. For coupling, it uses the average "relatedness" between all pairs of rules, where members of the pairs lie in different groups. For overall cohesion, it uses the average relatedness of every pair of rules that lie in the same group. For the battle system, these quantities are −0.2443 average coupling and +4.1631 average cohesion, suggesting, at the least, a far better than random organization.

CONCLUSIONS

By studying the connectivity of rules and facts in rule-based expert systems, we find that they indeed have a latent structure that can be used to support a new programming methodology. We have developed a method based on dividing the rules into groups and concentrating on these facts that carry information between rules in any two different groups. We have developed an algorithm for grouping the rules of a knowledge base automatically and a simple notation and set of software tools to support the new method.

The resulting programming method requires the knowledge engineer who develops a rule-based system to declare groups of rules, flag all between-group facts, and provide descriptions of those facts to any rule groups that use such facts. The knowledge engineer who wants to modify such a system then gives special attention to the between-group facts and preserves or relies on their descriptions when making changes. In contrast to the homogeneous way in which the facts of a rule-based system are usually viewed, this method distinguishes certain facts as more important than others and directs the programmer's attention to them.

A future step in this research will be to attempt to measure the extent to which the new method helps or hinders maintenance of an expert system. We will attempt to make changes both to a conventional expert system and to one divided into groups following the proposed method and compare the results. A second future direction will be to apply these basic ideas to newer knowledge representations, such as frames and semantic nets, as large systems begin to be written using them. The basic approach will likely remain the same: divide the information in the knowledge base into groups, then specify and limit the flow of information between the groups. The specifics of the programming method will, of course, differ.

ACKNOWLEDGMENTS

We are most grateful to several artificial intelligence researchers who made knowledge bases they developed available to us: Ralph Fink of the Naval Air Development Center and James Reggia of the University of Maryland developed the systems discussed in this report. Anne Werkheiser of the Army Engineer Topographic Laboratory and Mark Lerner of Columbia University also contributed their systems to this work. We also thank David Parnas for his comments on an earlier version of this report.

REFERENCES

1. D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM* **15**, 1053-1058 (1972).

2. D.L. Parnas, "Software Engineering Principles," *INFOR Can. J. Oper. Res. Inf. Proc.* **22**, 303-316 (1984).

3. J. McDermott, "R1: The Formative Years," *The AI Magazine,* **21-29** (1981).

4. J.B. Wright, F.D. Miller, G.V.E. Otto, E.M. Siegfried, G.T. Vesonder, and J.E. Zielinski, "ACE: Going from Prototype to Product with an Expert System," *Proc. 1984 ACM Annual Conf. 5th Generation Challenge,* **24-28**, 1984.

5. R.O. Duda, P.E. Hart, N.J. Nilsson, and G.L. Sutherland, "Semantic Network Representations in Rule-based Inference Systems," pp. **203-221** in *Pattern-directed Inference Systems,* ed. D.A. Waterman and F. Hayes-Roth (Academic Press, New York 1978).

6. D.G. Bobrow and M. Stefik, "The LOOPS Manual," Tech. Rep. KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center (1981).

7. W.J. Clancey, "The Advantages of Abstract Control Knowledge in Expert System Design," *Proc. National Conference on Artificial Intelligence (1983), 74-78.*

8. M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, "The Architecture of Expert Systems," pp. **89-126** in *Building Expert Systems*, ed. F. Hayes-Roth, E.A. Waterman, and D.B. Lenat (Addison-Wesley, Reading, Mass., 1983).

9. A. Gupta and C.L. Forgy, "Measurements on Production Systems," CMU-CS-83-167, Computer Science Department, Carnegie-Mellon University (1983).

10. P.H. Winston and B.K.P. Horn, *LISP* (Addison-Wesley, Reading, Mass., 1980).

11. R.J.K. Jacob and J.N. Froscher, "Developing a Software Engineering Methodology for Rule-based Systems," *1985 Conference on Intelligent Systems and Machines,* Oakland University (1985).

12. R.J.K. Jacob and J.N. Froscher, "Software Engineering for Rule-based Systems," *Proc. Fall Joint Computer Conference*, Dallas Tex. (1986) in press.

13. L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming* (Addison-Wesley, Reading, Mass., 1985).

14. C.L. Forgy, "OPS5 User's Manual," Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University (1981).

15. J. Reggia, "Knowledge-based Decision Support Systems: Development through KMS," *Department of Computer Science, University of Maryland* (1981).

16. W.P. Stevens, G.J. Meyers, and L.L. Constantine, "Structured Design," *IBM Systems Journal* **13**, 115-139 (1974).